

Performance analysis of the Kahan-enhanced scalar product on current multicore processors

J. Hofmann¹, D. Fey¹, J. Eitzinger², G. Hager², and G. Wellein²

¹Chair for Computer Architecture, University Erlangen-Nuremberg

²Erlangen Regional Computing Center (RRZE), University Erlangen-Nuremberg

Abstract. We investigate the performance characteristics of a numerically enhanced scalar product (dot) kernel loop that uses the Kahan algorithm to compensate for numerical errors, and describe efficient SIMD-vectorized implementations on recent Intel processors. Using low-level instruction analysis and the execution-cache-memory (ECM) performance model we pinpoint the relevant performance bottlenecks for single-core and thread-parallel execution, and predict performance and saturation behavior. We show that the Kahan-enhanced scalar product comes at almost no additional cost compared to the naive (non-Kahan) scalar product if appropriate low-level optimizations, notably SIMD vectorization and unrolling, are applied. We also investigate the impact of architectural changes across four generations of Intel Xeon processors.

1 Introduction and related work

Accumulating finite-precision floating-point numbers in a scalar variable is a common operation in computational science and engineering. The consequences in terms of accuracy are inherent to the number representation and have been well known and studied for a long time [1]. There is a number of summation algorithms that enhance accuracy while maintaining an acceptable throughput [2, 3], of which Kahan [4] is probably the most popular one. However, the topic is still subject to active research [5–8]. A straightforward solution to the inherent accuracy problems is arbitrary-precision floating point arithmetic, which comes at a significant performance penalty. Naive summation and arbitrary precision arithmetic are at opposite ends of a broad spectrum of options, and balancing performance vs. accuracy is a key concern when selecting a specific solution.

Naive summation, which simply adds each successive number in sequence to an accumulator, requires appropriate unrolling for SIMD vectorization and pipelining. The necessary code transformations are performed automatically by modern compilers, which results in optimal in-core performance. Such a code quickly saturates the memory bandwidth of modern multi-core CPUs when the data is in memory.

This paper investigates implementations of the scalar product, a kernel which is relevant in many numerical algorithms. Starting from an optimal naive implementation it considers scalar and SIMD-vectorized versions of the Kahan algorithm using various SIMD instruction set extensions on a range of current Intel processors. Using an analytic performance model we point out the conditions under which Kahan comes for free, and we predict the single core performance in all memory hierarchy levels as well as the scaling behavior across the cores of a chip.

2 Performance modeling on the core and chip level

The ECM model [9–11] is an extension of the well-known Roofline model [12]. It estimates the number of CPU cycles required to execute a number of iterations n_{it} of a loop on a single core of a multicore chip. It considers the time for executing the iterations with data coming from the L1 cache as well as the time for moving the required cache lines (CLs) through the cache hierarchy. In the following we will assume fully inclusive caches, which is appropriate for current Intel architectures. We give a brief overview of the model here; details can be found in [11].

The ECM model considers the time to execute the instructions of a loop kernel on the processor core, assuming that there are no cache misses, and the time to transfer data between its initial location and the L1 cache. The in-core execution time T_{core} is determined by the unit that takes the most cycles to execute the instructions. Since data transfers in the memory hierarchy occur in units of cache lines (CLs), we always consider one cache line’s “worth of work.” E.g., with a loop kernel that handles single-precision floating-point arrays with unit stride, one unit of work is $n_{it} = 16$ iterations.

The time needed for all data transfers required to execute one work unit is the “transfer time.” We neglect all latency effects, so the cost for one CL transfer is set by the maximum bandwidth. E.g., on the Intel IvyBridge architecture, one CL transfer takes two cycles between adjacent cache levels. Getting a 64-byte CL from memory to L3 or back takes $64 \text{ bytes} \cdot f / b_S$ cycles, where f is the CPU clock speed and b_S is the memory bandwidth. Note that in practice we encounter the problem that the model is too optimistic for in-memory data sets on some processors. This can be corrected by introducing a latency penalty. See Sect. 3 for details.

The in-core execution and transfer times must be put together to arrive at a prediction of single-thread execution time. If T_{data} is the transfer time, T_{OL} is the part of the core execution that overlaps with the transfer time, and T_{nOL} is the part that does not, then

$$T_{core} = \max(T_{nOL}, T_{OL}) \quad \text{and} \quad T_{ECM} = \max(T_{nOL} + T_{data}, T_{OL}) . \quad (1)$$

The model assumes that (i) core cycles in which loads are retired do not overlap with any other data transfer in the memory hierarchy, but all other in-core cycles (including pipeline bubbles) do, and (ii) the transfer times up to the L1 cache are mutually non-overlapping. A shorthand notation is used to summarize the relevant information about the cycle times that comprise the model for a loop: We write the model as $\{T_{OL} \parallel T_{nOL} \mid T_{L1L2} \mid T_{L2L3} \mid T_{L3Mem}\}$, where T_{nOL} and T_{OL} are as defined above, and the other quantities are the data transfer times between adjacent memory hierarchy levels. Cycle predictions for data sets fitting into any given memory level can be calculated from this by adding up the appropriate contributions from T_{data} and T_{nOL} and applying (1). For instance, if the ECM model reads $\{2 \parallel 4 \mid 4 \mid 4 \mid 9\}$ cy, the prediction for L2 cache will be $\max(2, 4 + 4)$ cy = 8 cy. As a shorthand notation for predictions we use a similar format but with “ \mid ” as the delimiter. For the above example this would read as $T_{ECM} = \{4 \mid 8 \mid 12 \mid 21\}$ cy. Converting from time (cycles) to performance is done by dividing the work W (e.g., flops) by the runtime: $P = W / T_{ECM}$. If T_{ECM} is given in clock cycles but the desired unit of performance is F/s, we have to multiply by the clock speed.

Microarchitecture	SandyBridge-EP	IvyBridge-EP	Haswell-EP	Broadwell-D
Shorthand	SNB	IVB	HSW	BDW
Xeon Model	E5-2680	E5-2690 v2	E5-2695 v3	D-1540
Year	03/2012	09/2013	09/2014	03/2015
Clock speed (fixed)	2.7 GHz	2.2 GHz	2.3 GHz	1.8 GHz
Cores/Threads	8/16	10/20	14/28	8/16
Load/Store throughput per cycle				
AVX(2)	1 LD & 1/2 ST	1 LD & 1/2 ST	2 LD & 1 ST	2 LD & 1 ST
SSE/scalar	2 LD 1 LD & 1 ST	2 LD 1 LD & 1 ST	2 LD & 1 ST	2 LD & 1 ST
L1 port width	2×16+1×16 B	2×16+1×16 B	2×32+1×32 B	2×32+1×32 B
ADD throughput	1 / cy	1 / cy	1 / cy	1 / cy
MUL throughput	1 / cy	1 / cy	2 / cy	2 / cy
FMA throughput	n/a	n/a	2 / cy	2 / cy
L2-L1 data bus	32 B	32 B	64 B	64 B
L3-L2 data bus	32 B	32 B	32 B	32 B
LLC size	20 MiB	25 MiB	35 MiB	12 MiB
Main memory	4×DDR3-1600	4×DDR3-1866	4×DDR4-2133	4×DDR4-2133
Peak memory BW	51.2 GB/s	51.2 GB/s	68.3 GB/s	34.1 GB/s
Load-only BW	43.6 GB/s (85%)	46.1 GB/s (90%)	60.6 GB/s (89%)	33 GB/s (95%)
T_{L3Mem} per CL	3.96 cy	3.05 cy	2.43 cy	3.49 cy

Table 1: Test machine specifications and micro-architectural features (one socket). The cache line length is 64 bytes in all cases. The SIMD register width is 16 bytes for SSE and 32 bytes for AVX.

We assume that the single-core performance scales linearly until a bottleneck is hit. On modern Intel processors the only bottleneck is the memory bandwidth, which means that an upper performance limit is given by the Roofline prediction for memory-bound execution: $P_{BW} = I \cdot b_S$, where I is the computational intensity of the loop code. The performance scaling for n cores is thus described by $P(n) = \min(n P_{ECM}^{mem}, I \cdot b_S)$ if P_{ECM}^{mem} is the ECM model prediction for data in main memory. The performance will saturate at $n_S = \lceil T_{ECM}^{mem} / T_{L3Mem} \rceil$ cores. In the following section we will use the ECM model to describe performance properties of different dot implementations.

3 Optimal implementations and performance models for dot

Table 1 gives an overview of the relevant architectural details of the four generations of Intel Xeon processors used in this work. The CPUs were released in successive years between 2012 and 2015. Intel Haswell-EP marks the big micro-architectural change, with a new SIMD instruction set extension (AVX2) and several fused multiply-add instructions (FMA3). There are also notable improvements in the memory hierarchy: The access path width of load/store units was widened from 16 bytes to 32 bytes, and the bus width between the L2 and the L1 cache was enlarged from 32 bytes to 64 bytes.

<p>(a)</p> <pre> float sum = 0.0; for (int i=0; i<n; i++) { sum = sum + a[i] * b[i] } </pre>	<p>(b)</p> <pre> float sum = 0.0; float c = 0.0; for (int i=0; i<N; ++i) { float prod = a[i]*b[i]; float y = prod-c; float t = sum+y; c = (t-sum)-y; sum = t; } </pre>
--	---

Fig. 1: (a) Naive scalar product code in single precision. (b) Kahan-compensated scalar product code.

The Broadwell chip is a very recent power-efficient “Xeon D” variant. All results for Broadwell are preliminary since we only had access to a pre-release version of the chip.

We first discuss variants for dot in single precision (SP) for the Intel IvyBridge microarchitecture. The differences to double precision (DP) and the impact of architectural changes are covered in Sect. 3. To eliminate variations introduced by compilers we implemented all kernels directly in assembly language using the `likwid-bench` microbenchmarking framework [13].

Naive scalar product The naive scalar product in single precision serves as the baseline (see Fig. 1a). Sufficient unrolling must be applied to hide the ADD pipeline latency for the recursive update on the accumulation register and to apply SIMD vectorization. Both optimizations introduce partial sums and are therefore not compatible with the C standard as the order of non-associative operations is changed. With higher optimization levels the current Intel compiler (version 15.0.2) is able to generate optimal code. Note that partial sums usually improve the accuracy of the result [8].

This kernel is limited by the throughput of the LOAD unit on the IVB architecture (see Table 1). Two AVX loads per vector (a and b) are required to cover one unit of work (16 scalar loop iterations), resulting in $T_{nOL} = 4$ cy. The overlapping part is $T_{nOL} = 2$ cy since two MULT and two ADD instructions must be executed. Data transfers between cache levels require two cycles per CL, so that $T_{L1L2} = T_{L2L3} = 4$ cy.

For T_{L3Mem} we calculate the number of cycles per CL transfer from the maximum memory bandwidth and the clock speed (last row in Table 1) and arrive at $T_{L3Mem} = 6.1$ cy. The full ECM model thus reads $\{2 \parallel 4 \mid 4 \mid 6.1\}$ cy. On newer Intel chips (notably IVB and HSW) unknown peculiarities in the design of the Uncore lead to extra latency penalties per cache line from memory. We take these deviations into account by introducing a penalty parameter that is fixed empirically. This parameter is an additive contribution to T_{L3Mem} , so that the final model is $\{2 \parallel 4 \mid 4 \mid 6.1 + 2.9\}$ cy, leading to a runtime prediction of $\{4 \mid 8 \mid 12 \mid 18.1 + 2.9\}$ cy. At a clock speed of 2.2 GHz the expected serial performance is thus

$$P = \frac{16 \text{ updates} \cdot 2.2 \text{ Gcy/s}}{\{4 \mid 8 \mid 12 \mid 18.1 + 2.9\} \text{ cy}} = \{8.80 \mid 4.40 \mid 2.93 \mid 1.68\} \text{ GUP/s} . \quad (2)$$

We choose an “update” (two flops) as the basic unit of work to make performance results for different implementations comparable. The predicted saturation point is at $n_S = \lceil (18.1 + 2.9)/6.1 \rceil = 4$ cores. Note that the maximum memory bandwidth has to be taken into account for the saturation point, so we divide by 6.1 cy. The Roofline “light speed,” i.e., the memory bandwidth-limited saturated performance, can be calculated from the computational intensity of one update per eight bytes: $P_{BW} = (1 \text{ update}/8B) \cdot b_S = 5.76 \text{ GUP/s}$.

All versions of the enhanced scalar product described in the next section will be compared to the optimal naive implementation.

Kahan-enhanced scalar product on IvyBridge Figure 1b shows the implementation of the Kahan algorithm for dot. Compilers have problems with this loop code for two reasons: First, the compiler detects (correctly) a loop-carried dependency on c , which prohibits SIMD vectorization and modulo unrolling. Second, the compiler may recognize that, arithmetically, c is always equal to zero. With high optimization levels it may thus reduce the code to the naive scalar product, defeating the purpose of the Kahan algorithm. This is the reason why we use hand-coded assembly throughout this work.

One iteration comprises one multiplication, four additions or subtractions, and two loads. The bottleneck on the IVB core level is thus the ADD unit (ADD and SUB are handled by the same pipeline). In the following we construct the ECM model for scalar, SSE, and AVX versions of the Kahan loop. Independent of vectorization we always establish proper modulo unrolling for best pipeline utilization.

Scalar implementation. In scalar mode, one unit of work amounts to $16 \times 4 = 64$ instructions in the ADD unit, resulting in $T_{OL} = 64 \text{ cy}$. Since two scalar loads can be executed per cycle on the IVB core, the 32 loads lead to $T_{nOL} = 16 \text{ cy}$. The contributions from in-cache and memory transfers are the same as for the naive variant above, so the complete ECM model is $\{64 \parallel 16 \mid 4 \mid 4 \mid 6.1 + 2.9\} \text{ cy}$, and the runtime prediction is $\{64 \parallel 64 \parallel 64 \parallel 64\} \text{ cy}$. According to the model the scalar variant should not be able to saturate the memory bandwidth using all cores on the ten-core chip, since $n_S = \lceil 64/6.1 \rceil = 11$ cores. The analysis shows that the scalar variant of Kahan is limited by the instruction throughput, specifically on the ADD pipeline, regardless of where the data resides. We thus expect the same performance $P = 16 \cdot 2.2/64 \text{ GUP/s} = 0.55 \text{ GUP/s}$ in all memory hierarchy levels for single-threaded execution, and close to perfect scalability across the cores of the chip.

SSE implementation. SSE uses 16-byte wide registers, and all instructions required for the Kahan algorithm exist in SSE variants, so the overall number of instructions is reduced by a factor of four compared to the scalar version, but the same throughput limits apply for the ADD and the LOAD unit. This leads to an ECM model of $\{16 \parallel 4 \mid 4 \mid 4 \mid 6.1 + 2.9\} \text{ cy}$ and a prediction of $\{16 \parallel 16 \parallel 16 \parallel 18.1 + 2.9\} \text{ cy}$, which yields $P = \{2.20 \parallel 2.20 \parallel 2.20 \parallel 1.68\} \text{ GUP/s}$. The SSE code is limited by the instruction throughput up to the L3 cache since all data transfer contributions can be overlapped with the ADD instructions. The optimal $4\times$ speed-up of SSE is thus observed in this case. For data in main memory the speed-up is just about $64/21 \approx 3\times$, and the single-core performance and saturation behavior are identical to the naive scalar product.

	ECM model [cy]	Prediction [cy/CL]	Pred. performance [GUP/s]
SNB	$\{8 \parallel 4 \mid 4 \mid 7.9 + 5.1\}$	$\{8 \parallel 8 \parallel 12 \parallel 19.9 + 5.1\}$	$\{5.40 \parallel 5.40 \parallel 3.60 \parallel 1.73\}$
IVB	$\{8 \parallel 4 \mid 4 \mid 6.1 + 2.9\}$	$\{8 \parallel 8 \parallel 12 \parallel 18.1 + 2.9\}$	$\{4.40 \parallel 4.40 \parallel 2.93 \parallel 1.68\}$
HSW	$\{8 \parallel \mathbf{2} \mid \mathbf{2} \mid 5.54 \mid 4.9 + 11.1\}$	$\{8 \parallel 8 \parallel 9.54 \parallel 14.44 + 11.1\}$	$\{4.60 \parallel 4.60 \parallel 3.86 \parallel 1.44\}$
BDW	$\{8 \parallel 2 \mid 2 \mid \mathbf{4} \mid 7 + \mathbf{1}\}$	$\{8 \parallel 8 \parallel 8 \parallel 15 + 1\}$	$\{3.60 \parallel 3.60 \parallel 3.60 \parallel 1.8\}$

Table 2: Comparison of the ECM model for optimal AVX implementations across the multicore Xeon CPUs in the testbed (see Table 1). The consequences of relevant architectural changes to the preceding generation are highlighted.

AVX implementation. AVX further reduces the runtime for the ADD operations by a factor of two, so $T_{OL} = 8$ cy. Although the number of LOAD instructions is also cut in half, the non-overlapping time T_{nOL} does not change, because the two LOAD ports of the L1 cache are only 16 bytes wide. Therefore only one LOAD instruction can be retired per cycle. The complete ECM model is $\{8 \parallel 4 \mid 4 \mid 6.1 + 2.9\}$ cy, the runtime prediction is $\{8 \parallel 8 \parallel 12 \parallel 18.1 + 2.9\}$ cy (leading to $P = \{4.40 \parallel 4.40 \parallel 2.93 \parallel 1.68\}$ GUP/s), and the saturation behavior is the same as for the SSE variant of Kahan and the naive scalar product. The AVX code is limited by the instruction throughput up to the L2 cache, and the full $2\times$ advantage versus SSE can be observed in this case. Starting from L3 there is a slight impact on runtime by data transfers, leading to a reduced speed-up of $1.3\times$ in L3 and none at all in main memory. Again the saturation behavior is expected between three and four cores.

The conclusion from this analysis is that there is no expected performance difference for in-memory working sets between the naive scalar product and the Kahan version if any kind of vectorization is applied to Kahan. With AVX, Kahan comes for free even in the L3 or the L2 cache. Only for in-L1 data we expect a $2\times$ slowdown for Kahan versus the naive version even with the best possible code.

Influence of processor architecture In this section we compare the model-based analysis across four generations of Intel CPUs: SandyBridge-EP (SNB), IvyBridge-EP (IVB), Haswell-EP (HSW), and Broadwell (BDW, in a power-efficient “Xeon D” variant). This covers four Intel Xeon microarchitectures over a time of three years and involves one major architectural step (from IVB to HSW). We always consider the optimal AVX code for the comparisons. There is no major change expected between SNB and IVB, since no dot-relevant hardware features were added. All observed performance differences are thus rooted in the clock speed and memory bandwidth (first row in Table 2). Note that despite the lower memory bandwidth of the SNB test system compared to IVB, the in-memory performance is higher due to the faster clock speed of SNB. The HSW microarchitecture has new features which influence dot performance: It can sustain two AVX loads and one AVX store per cycle, effectively doubling LOAD-/STORE throughput. In addition, the L1-L2 bus width was doubled, allowing for a full CL transfer per cycle. These changes result in $T_{nOL} = 2$ cy and $T_{L1L2} = 2$ cy (third row in Table 2). Here we encounter the peculiarity that HSW lowers the Uncore clock speed if only a single core is used. This is the reason why the T_{L2L3} contribution is 5.54 cy

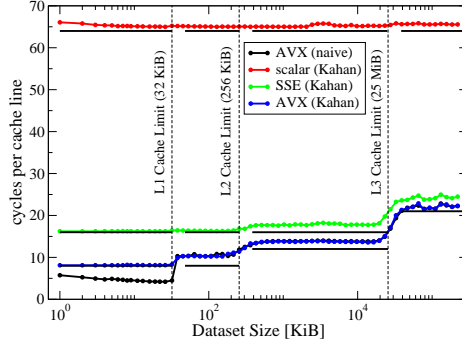


Fig. 2: Single-core cycles per CL vs. data set size for various implementations of the Kahan scalar product and the AVX version of the naive scalar product in SP on IVB. The horizontal lines represent the ECM model predictions for scalar (top), SSE (middle), and AVX (bottom) Kahan variants.

instead of 4cy. The BDW architecture has introduced no relevant changes, but it does not show the Uncore slowdown like HSW (fourth row in Table 2).¹ It is interesting that BDW only requires half a cycle of latency penalty per CL in memory, so the uncorrected ECM model works very well already. BDW performance is insensitive to data transfers up to the L3 cache.

Double vs. single precision The model prediction in terms of cycles per CL does not change for the SIMD variants of Kahan when going from SP to DP, but one CL update represents twice as much useful work (scalar iterations) in the SP case. However, the penalty for going from SIMD to scalar is only half as big as for SP, since the scalar register width is eight bytes instead of four. The ECM model for the DP scalar version of the Kahan dot on IVB is $\{32 \parallel 8 \mid 4 \mid 4 \mid 6.1 + 2.9\}$ cy and the according runtime prediction is $\{32 \mid 32 \mid 32 \mid 32\}$ cy, with $P = 0.55$ GUP/s. The reduced cycle count (32 instead of 64) for DP leads to saturation at a smaller number of cores for in-memory working sets: $n_S = \lceil 32/6.1 \rceil = 6$. Hence, even the scalar DP variant of Kahan exerts sufficient pressure on the memory interface to reach saturation. The saturated DP performance according to the Roofline model is $P_{BW} = (1 \text{ update}/16B) \cdot b_S = 2.88$ GUP/s.

4 Performance results and model validation

Single-core benchmarking results for single precision on IVB are shown in Fig. 2. The model predicts the overall behavior very well. The naive and the AVX Kahan version show identical performance in L2 cache and beyond. As predicted there is no performance drop for the SSE Kahan version from L1 to L2. Both AVX Kahan and the compiler-generated naive version fall slightly short of the prediction in L2. This is a general observation with many loop kernels, and we interpret it as a consequence of the L2-L1 hardware prefetcher doing a better job in latency hiding for SSE than for AVX due to the more relaxed timings in the SSE case. Since the details of prefetching are undisclosed, we have no way to prove or refute this hypothesis. Finally, the constant

¹ Note that our test system was a pre-release Xeon D; production systems and mainstream Xeon Broadwell chips may show a different behavior.

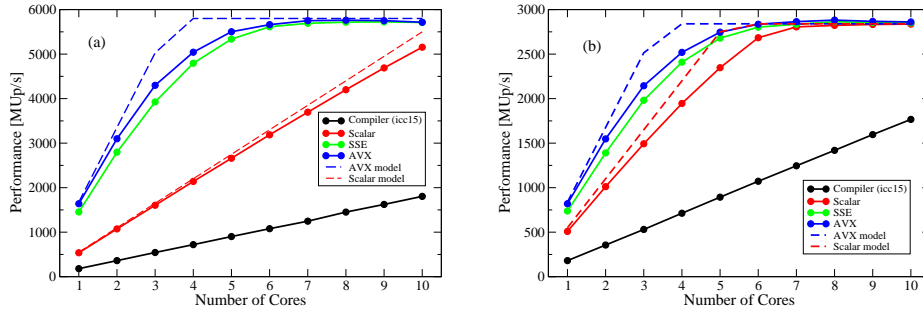


Fig. 3: In-memory scaling for different implementation of the Kahan scalar product on IVB for (a) single precision and (b) double precision. Model predictions are shown with dashed lines for the scalar and AVX versions.

performance of the scalar Kahan variant across all memory levels is perfectly predicted by the model.

In-memory scaling results on the chip level are shown in Fig. 3a. The dashed lines are the model predictions (for clarity we only show models for scalar and AVX). As anticipated via the ECM model, the scalar version cannot saturate the memory bandwidth even if all cores are used. Since any code that is able to saturate the bandwidth is “perfect,” any kind of vectorization will make the Kahan algorithm as fast as the naive scalar product. Note, however, that on a CPU with a faster clock speed or more cores saturation will be easily achieved even with scalar code. This effect illustrates the general observation that more parallelism can “heal” low single-core performance. For comparison we also show the compiler-generated variant of Kahan. As described earlier, the code is devastatingly slow since the compiler cannot resolve the loop-carried dependency.

The only relevant difference between DP and SP is the smaller performance penalty of the scalar variant for DP (see Fig. 3b), which leads to strong saturation at about six cores as predicted by the model.

In order to compare different architectures, we show single-core data for the AVX-vectorized Kahan scalar product in Fig. 4a. Since we report runtime in cycles per CL, the influence of different clock speeds and memory bandwidths is only visible for the in-memory case. In the L1 cache all processors show the same runtime, because the architectural improvements with HSW and BDW do not address the bottleneck of the algorithm at hand (the ADD throughput). In L2 and L3, HSW and BDW show higher performance than the previous generations due to their doubled L2-L1 bandwidth and LOAD throughput. The step from L2 to L3 is important, since it marks the transition to the Uncore, which is a shared resource across the cores. Although a notable improvement is seen in L3 with each new architecture, we observe efficiency issues in the Uncore on IVB and HSW that prevent those CPUs from attaining the expected performance. In memory, HSW is a significant step back in terms of single-core performance due to a large latency penalty. BDW seems to have corrected those issues, but we must stress again that these observations were made on an eight-core single-socket “Xeon D”

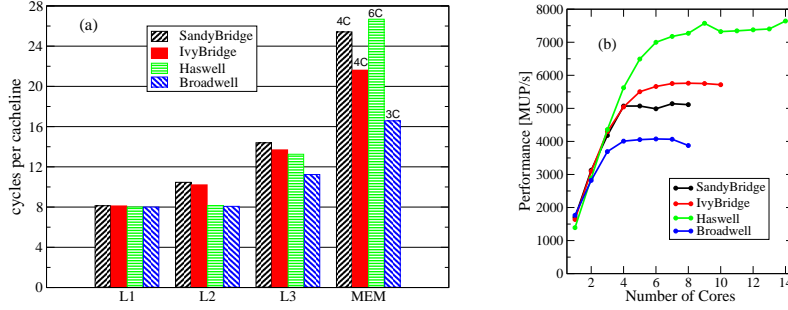


Fig. 4: Comparison between four Intel Xeon multi-core architectures using the single-precision AVX Kahan scalar product: (a) Measured single-core runtime in cycles per CL in different memory hierarchy levels. The saturation point n_s is indicated above the bars for the memory-bound case. (b) Measured performance scaling with in-memory working set.

chip, and it is unclear if the to be released multi-socket variants with larger core counts can live up to the expectations raised here. It is also worth emphasizing that, as already mentioned above, in practice any code that can saturate the memory bandwidth is “good enough.” Figure 4b shows the in-memory performance scaling for all four architectures with the AVX Kahan scalar product: The differences in saturated performance indeed reflect the differences in saturated memory bandwidth. Again, and certainly as expected from the model, vectorization makes the Kahan algorithm come for free.

There is one additional optimization on HSW and BDW that we have not mentioned yet. The two FMA units can theoretically increase the ADD throughput by a factor of two. Both units can execute FMA and MULT instructions, but only one of them can handle stand-alone ADDs. This is not a problem with hand-crafted assembly since one can endow an FMA instruction with a unit multiplicand to act like an ADD. The downside is that the FMA instruction has a higher latency of five cycles (ADD only has three) and therefore requires deeper unrolling to hide the pipeline latency. Both architectures hence run out of registers and only achieve a 20% speed-up from FMA with data in L1, and no noticeable improvement beyond L1.

5 Conclusion

We have investigated the performance of naive and Kahan-enhanced variants of the scalar product on a range of recent Intel multicore chips. Using the ECM model the single-core performance in all memory hierarchy levels and the multi-core scaling for in-memory data were accurately described. The most important result is that even the single-threaded Kahan algorithm comes with no performance penalties on all standard multicore architectures under investigation in the L2 cache, the L3 cache, and in memory if implemented optimally. Depending on the particular architecture and whether single or double precision is used, even scalar code may achieve bandwidth saturation in memory when using multiple threads. Performance improvements between successive generations of Intel CPUs could be attributed to specific architectural advance-

ments, such as increased LOAD throughput on Haswell or a more efficient Uncore on Broadwell.

We emphasize that the approach and insights described here for the special case of the Kahan scalar product can serve as a blueprint for other load-dominated streaming kernels.

Acknowledgement We thank Intel Germany for providing an early access Broadwell test system. This work was partially funded by BMBF under grant 01IH13009A (project FEPA), and by the Competence Network for Scientific High Performance Computing in Bavaria (KONWIHR).

References

1. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* **23**(1) (March 1991) 5–48
2. Linz, P.: Accurate floating-point summation. *Commun. ACM* **13**(6) (June 1970) 361–362
3. Gregory, J.: A comparison of floating point summation methods. *Commun. ACM* **15**(9) (September 1972) 838–
4. Kahan, W.: Pracniques: Further remarks on reducing truncation errors. *Commun. ACM* **8**(1) (January 1965) 40–
5. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput.* **31**(1) (October 2008) 189–224
6. Zhu, Y.K., Hayes, W.B.: Algorithm 908: Online exact summation of floating-point streams. *ACM Trans. Math. Softw.* **37**(3) (2010) 37:1–37:13
7. Demmel, J., Nguyen, H.D.: Fast reproducible floating-point summation. In: *Computer Arithmetic (ARITH)*, 2013 21st IEEE Symposium on. (April 2013) 163–172
8. Dalton, B., Wang, A., Blainey, B.: SIMDizing pairwise sums: A summation algorithm balancing accuracy with throughput. In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing. WPMVP '14*, New York, NY, USA, ACM (2014) 65–70
9. Treibig, J., Hager, G.: Introducing a performance model for bandwidth-limited loop kernels. In Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J., eds.: *Parallel Processing and Applied Mathematics. Volume 6067 of Lecture Notes in Computer Science.*, Springer Berlin / Heidelberg (2010) 615–624
10. Hager, G., Treibig, J., Habich, J., Wellein, G.: Exploring performance and power properties of modern multicore chips via simple machine models. *Concurrency Computat.: Pract. Exper.* (2013) DOI: 10.1002/cpe.3180.
11. Stengel, H., Treibig, J., Hager, G., Wellein, G.: Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. In: *Proceedings of the 29th ACM International Conference on Supercomputing. ICS '15*, New York, NY, USA, ACM (2015)
12. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* **52**(4) (2009) 65–76
13. Treibig, J., Hager, G., Wellein, G.: likwid-bench: An extensible microbenchmarking platform for x86 multicore compute nodes. In Brunst, H., et al., eds.: *Tools for High Performance Computing 2011*. Springer Berlin Heidelberg (2012) 27–36